Practical Power Analysis Attacks on Software Implementations of McEliece

Stefan Heyse, Amir Moradi, and Christof Paar

Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{heyse, moradi, cpaar}@crypto.rub.de

Abstract. The McEliece public-key cryptosystem is based on the fact that decoding unknown linear binary codes is an NP-complete problem. The interest on implementing post-quantum cryptographic algorithms, e.g. McEliece, on microprocessor-based platforms has been extremely raised due to the increasing storage space of these platforms. Therefore, their vulnerability and robustness against physical attacks, e.g., state-of-the-art power analysis attacks, must be investigated. In this work, we address mainly two power analysis attacks on various implementations of McEliece on an 8-bit AVR microprocessor. To the best of our knowledge, this is the first time that such side-channel attacks are practically evaluated.

1 Introduction

1.1 Motivation

Mainly all modern security systems rely on public-key cryptography. Most commonly used are RSA, ECC and Diffie-Hellman (DH) based schemes. They are well understood and withstand many types of attacks for years. However, these cryptosystems rely on two primitive security assumptions, namely the factoring problem (FP) and the discrete logarithm problem (DLP). A breakthrough in one of the currently known attacks (e.g, *Number Field Sieve* or *Index Calculus*) or newly introduced successful build of a quantum computer can make all of them useless.

Fortunately, there are alternative public-key primitives, like hash-based, lattice-based, MQ-methods and coding-based schemes. During the last years, much research effort has been put into determining strengths and weaknesses of these systems. They were implemented on different platforms, e.g., x86- and x64-based CPUs, GPUs, FPGAs and small embedded systems employing microcontrollers.

The oldest scheme, which is based on coding theory, has been presented by Robert J. McEliece in 1978. In its original form it withstands all the attacks. The most recent and effective attack reported during the last 30 years reduces the security of a system from 80- to 60-bit [2]. It has been implemented on several platforms including CPU [3], GPU [13], FPGA [8, 28], and 8-bit microcontrollers [8]. To make this system a real alternative to the existing schemes,

or to let it be a candidate for the post-quantum era, all possible attacks have to be evaluated.

Since the first introduction of DPA in 1999 [15], it has become an alternative approach for extracting secret key of cryptographic devices by exploiting side-channel leakages despite robustness of the corresponding algorithms to cryptanalyzing methods. During the last ten years it has been showed by many articles (cf., e.g., the CHES workshop proceedings since 1999) that side-channel attacks must be considered as a potential risk for security related devices. The practical attacks mounted on a real-world and widespread security applications, e.g., [9], also strengthen the importance of side-channel attacks.

Though there are a few works on vulnerability and robustness of implementation of public-key algorithms to side-channel attacks (e.g., [7, 20]), most of the published articles in the scientific literatures on this field concentrated on the state-of-the-art Differential Power Analysis (DPA) attacks on symmetric block ciphers, e.g., [21, 25, 30]. In addition to classical DPA approaches (like [5, 11, 15, 19]) combination of cryptanalytic schemes and side-channel leakages led to innovative attacks, e.g., a couple of collision side-channel attacks [4, 24, 26] and algebraic side-channel attacks [23].

1.2 Related Works and Our Contribution

There are not many articles regarding the evaluation of side-channel attacks on post-quantum algorithms. The lattice based NTRUencrypt has been investigated in [29] and [34]. Hash trees look more like a protocol than to a cryptographic primitive, and they stand or fall with the underlying hash function. For MQ-based algorithms, we know no research inquiry. Only coding-based cryptography got some attention during the last two years.

Side-channel attacks on PC implementations of the McEliece scheme are already addressed in [32] and [12] where the authors mounted a timing attack. By means of this attack, the adversary would be able to decrypt only the attacked message. Though the attack is promising, the adversary needs to repeat the attack for every later ciphertext by having physical access to the target device. Further, recently another timing attack on McEliece has been published in [27] that shows the interest of the research community to side-channel attack on McEliece implementations.

Resistance of McEliece against fault injection attacks has been investigated in [6]. The authors stated that due to the error correction capability of this type of cryptosystem, it is heavily resistant against fault injection attacks because the faults are part of the algorithm and are simply corrected (maybe to a wrong message, but no secret information is revealed).

An algebraic side-channel attack on AES presented in [23] is able to recover the secret key of a microcontroller-based implementation by means of profiling and a single mean trace supposing that the attacker knows the execution path and can recover the Hamming weight (HW) of the operand of the selected instructions. Though this attack is even efficient to overcome arithmetic masking schemes supposing the same adversary model, solving the algebraic system

equations encounters many problems by wrong HW predictions. Our proposed attacks are partially based on the same idea, i.e., examining the secret hypotheses considering the predicted HW of the processed data.

In contrary to the side-channel attacks proposed on the McEliece implementations so far, we have implemented and practically evaluated all steps of our proposed attacks. The target implementations which are considered in this work are based on the article recently published in CHES 2009 [8]. Since there is not a unique way to implement the McEliece decryption scheme by a microcontroller, we define four different implementation profiles to realize the decryption algorithm, and for each of which we propose an attack to recover the secret key. Generally our proposed attacks can be divided into two phases:

- collecting side-channel observations for chosen ciphertexts and generating a candidate list for each target secret element of the cipher and
- examining the candidates to check which hypotheses match to the public parameters of the cipher.

By means of our proposed attacks we are able to recover the permutation matrix and the parity check matrix if each one is performed solely. On the other hand if both matrices are combined in the target implementation, we are also able to recover the combined (permutation and parity check) matrix. Each of these attacks leads to breaking the decryption scheme and recovering the secret key. It should be noted that contrary to the attack presented in [23] our supposed adversary model does not need to profile the side-channel leakage of the target device, and our proposed attacks are more insensitive to wrong HW predictions than that of presented in [23].

1.3 Organization

In the next section, a short introduction to McEliece cryptosystem is given. Then, in Section 3 the implementation profiles which are considered in our attacks as the target implementation are defined. Section 4 briefly reviews the concept of power analysis attacks. In Section 5 first our supposed adversary model is introduced. Then, we explain how to use side-channel observations to directly recover a secret (e.g., the permutation matrix) or to partially predict a part of a secret (e.g., the parity check matrix). Afterwards, we show how to break the system using the revealed/predicted information. Further, we discuss possible countermeasures to defeat our proposed attacks in Section 6. Finally, Section 7 concludes our research.

2 McEliece in a Flash

This section gives a short overview on the original McEliece cryptosystem, and introduces the used Goppa codes. We stay superficial, and explain only what is necessary to understand the attacks described afterwards.

Algorithm 1 McEliece Message Encryption

Require: $m, K_{pub} = (\hat{G}, t)$
Ensure: Ciphertext c
1: Encode the message m as a binary string of length k
2: $c' \leftarrow m \cdot \hat{G}$
3: Generate a random n-bit error vector z containing at most t ones
4: $c = c' + z$
5: **return** c

Algorithm 2 McEliece Message Decryption

Require: $c, K_{sec} = (P^{-1}, G, S^{-1})$
Ensure: Plaintext m
1: $\hat{c} \leftarrow c \cdot P^{-1}$
2: Use a decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m} = m \cdot S$
3: $m \leftarrow \hat{m} \cdot S^{-1}$
4: **return** m

2.1 Background on the McEliece Cryptosystem

The McEliece scheme is a public-key cryptosystem based on linear error-correcting codes proposed by Robert J. McEliece in 1978 [18]. The secret key is an efficient decoding algorithm of an error-correcting code with dimension k, length n and error correcting capability t. To create a public key, McEliece defines a random $k \times k$-dimensional scrambling matrix S and $n \times n$-dimensional permutation matrix P disguising the structure of the code by computing the product $\hat{G} = S \times G \times P$, where G is the generator matrix of the code. Using the public key $K_{pub} = (\hat{G}, t)$ and private key $K_{sec} = (P^{-1}, G, S^{-1})$, encryption and decryption algorithms can be given by Algorithm 1 and Algorithm 2 respectively.

Note that Algorithm 1 only consists of a simple matrix multiplication with the input message and then distributes t random errors on the resulting code word.

Decoding the ciphertext c for decryption as shown in Algorithm 2 is the most time-consuming process and requires several more complex operations in binary extension fields. In Section 2.2 we briefly introduce the required steps for decoding codewords.

2.2 Classical Goppa Codes

This section reviews the underlying code-based part of McEliece without the cryptographic portion. To encode a message m into a codeword c, the message m should be represented as a binary string of length k and be multiplied by the $k \times n$ generator matrix G of the code. Decoding a codeword $\underline{r}$ at the receiver side with a (possibly) additive error vector $\underline{e}$ is much more complex than a simple matrix vector multiplication for encoding. The most widely used decoding scheme for Goppa codes is the Patterson algorithm [22].

Here we only give a short introduction and define the necessary abbreviations.

Theorem 1. *[33] Let $g(z)$ be an irreducible polynomial of degree t over $GF(2^m)$. Then the set*

$$\Gamma(g(z), GF(2^m)) = \{(c_\alpha)_{\alpha \in GF(2^m)} \in \{0,1\}^n \mid \sum_{\alpha \in GF(2^m)} \frac{c_\alpha}{z-\alpha} \equiv 0 \bmod g(z)\} \tag{1}$$

defines a binary Goppa code C of length $n = 2^m$, dimension $k \geq n - mt$ and minimum distance $d \geq 2t+1$. The set of the α_i is called the support $\mathcal{L}$ of the code.

This code is capable of correcting up to t errors [1] and can be described as a $k \times n$ generator matrix G such that $C = \{mG : m \in GF_2^k\}$. This matrix is systematic, if it is in the form $(I_k \| Q)$, where I_k denotes the $k \times k$ identity matrix and Q is a $k \times (n-k)$ matrix. Then $H = (Q^T \| I_{n-k})$ is a parity-check matrix of C with $C = \{c \in GF_2^n : cH^T = 0\}$.

Since $\underline{r} = \underline{c} + \underline{e} \equiv \underline{e} \mod g(z)$ holds, the syndrome $Syn(z)$ of a received codeword can be obtained from Equation (1) by

$$Syn(z) = \sum_{\alpha \in GF(2^m)} \frac{r_\alpha}{z-\alpha} \equiv \sum_{\alpha \in GF(2^m)} \frac{e_\alpha}{z-\alpha} \mod g(z) \tag{2}$$

To finally recover $\underline{e}$, we need to solve the key equation $\sigma(z) \cdot Syn(z) \equiv \omega(z) \mod g(z)$, where $\sigma(z)$ denotes a corresponding error-locator polynomial and $\omega(z)$ denotes an error-weight polynomial.

The roots of $\sigma(z)$ denote the positions of error bits. If $\sigma(\alpha_i) \equiv 0 \mod g(z)$ where α_i is the corresponding bit of a generator in $GF(2^m)$, there was an error in the position i of the received codeword that can be corrected by bit-flipping.

This decoding process, as required in Step 2 of Algorithm 2 for message decryption, is finally summarized in Algorithm 3 in the appendix.

Instead of writing inverted polynomials to the columns parity check matrix H, there exist an alternative representation for the parity check matrix, which is important for the attack in Section 5.3.

From Equation (2) we can derive the parity check matrix H as

$$H = \begin{Bmatrix} \frac{g_t}{g(\alpha_0)} & \frac{g_t}{g(\alpha_1)} & \cdots & \frac{g_t}{g(\alpha_{n-1})} \\ \frac{g_{t-1}+g_t\cdot\alpha_0}{g(\alpha_0)} & \frac{g_{t-1}+g_t\cdot\alpha_0}{g(\alpha_1)} & \cdots & \frac{g_{t-1}+g_t\cdot\alpha_0}{g(\alpha_{n-1})} \\ \vdots & \ddots & & \vdots \\ \frac{g_1+g_2\cdot\alpha_0+\cdots+g_t\cdot\alpha_0^{s-1}}{g(\alpha_0)} & \frac{g_1+g_2\cdot\alpha_0+\cdots+g_t\cdot\alpha_0^{s-1}}{g(\alpha_1)} & \cdots & \frac{g_1+g_2\cdot\alpha_0+\cdots+g_t\cdot\alpha_0^{s-1}}{g(\alpha_{n-1})} \end{Bmatrix} \tag{3}$$

This can be split into

$$H = \begin{Bmatrix} g_t & 0 & \cdots & 0 \\ g_{s-1} & g_t & \cdots & 0 \\ \vdots & \ddots & & \vdots \\ g_1 & g_2 & \cdots & g_t \end{Bmatrix} * \begin{Bmatrix} \frac{1}{g(\alpha_0)} & \frac{1}{g(\alpha_1)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\ \frac{\alpha_0}{g(\alpha_0)} & \frac{\alpha_1}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}}{g(\alpha_{n-1})} \\ \vdots & \ddots & & \vdots \\ \frac{\alpha_0^{s-1}}{g(\alpha_0)} & \frac{\alpha_1^{s-1}}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^{s-1}}{g(\alpha_{n-1})} \end{Bmatrix}, \tag{4}$$

where the first part has a non-zero determinant, and following the second part $\hat{H}$ is equivalent to the parity check matrix, which has a simpler structure. By applying the Gaussian algorithm to the second matrix $\hat{H}$ one can bring it to systematic form $(I_k \mid \underline{H})$, where I_k is the $k \times k$ identity matrix. Note that whenever a column swap is performed, a swap on the corresponding elements of the support $\mathcal{L}$ is also performed. From the systematic parity check matrix $(I_k \mid \underline{H})$, now the systematic generator matrix G can be derived as $(I_{n-k} \mid \underline{H}^{\mathrm{T}})$.

In the context of McEliece decryption, reverting the permutation can be merged into the parity check matrix by permuting the support $\mathcal{L}$. Using $\mathcal{L}_P = P^{-1} * \mathcal{L}$ to generate H leads to a parity check matrix that computes the correct syndrome for a permuted codeword.

In the following we always refer to a binary irreducible Goppa code with $m = 11$ and $t = 27$. This is a symmetric equivalent security of 80 bits and leads to $n = 2048$ and $k = 1751$ [2].

3 Practical Aspects

The combination of the McEliece decryption Algorithm 2 and the Goppa decoding Algorithm 3 allows a wide range of different implementations. For our proposed attacks, the most interesting point is the specific implementation of step 1 of Algorithm 2 and step 1 of Algorithm 3 and whether they are merged together or not. According to these points we define four so-called implementation profiles:

Profile I performs the permutation of the ciphertext and computes the columns of H as they are needed by either using the extended euclidean algorithm (EEA) or the structure given in Equation (3) or (4).

Profile II also performs the permutation, but uses the precomputed parity check matrix H.

Profile III does not really perform the permutation, but directly uses a permuted parity check matrix. As stated in Section 2.2, we can use $\mathcal{L}_P = P^{-1} * \mathcal{L}$ to compute the syndrome of the unpermuted ciphertext. This profile computes the permuted columns as needed.

Profile IV does the same as profile III, but uses a precomputed and permuted parity check matrix.

4 Introduction to Power Analysis Attacks

Power analysis attacks exploit the fact that the execution of a cryptographic algorithm on a physical device leaks information about the processed data and/or executed operations through instantaneous power consumption [15]. Measuring and evaluating the power consumption of a cryptographic device allows exploiting information-dependent leakage combined with the knowledge about the plaintext or ciphertext in order to extract, e.g., a secret key. Since intermediate result of the computations are serially processed (especially in 8-,16-, or 32-bit

architectures, e.g., general-purpose microcontrollers) a divide-and-conquer strategy becomes possible, i.e., the secret key could be recovered byte by byte.

A Simple Power Analysis (SPA) attack, as introduced in [15], relies on visual inspection of power traces, e.g., measured from an embedded microcontroller of a smartcard. The aim of an SPA is to reveal details about the execution of the program flow of a software implementation, like the detection of conditional branches depending on secret information. Recovering an RSA private key bit-by-bit by an SPA on square-and-multiply algorithm [15] and revealing a KeeLoq secret key by SPA on software implementation of the decryption algorithm [14] are amongst the powerful practical examples of SPA on real-world applications. Contrary to SPA, Differential Power Analysis (DPA) utilizes statistical methods and evaluates several power traces. A DPA requires no knowledge about the concrete implementation of the cipher and can hence be applied to most of unprotected black box implementations. According to intermediate values depending on key hypotheses the traces are correlated to estimated power values, and then correlation coefficients indicate the most probable hypothesis amongst all partially guessed key hypotheses [5]. In order to perform a correlation-based DPA, the power consumption of the device under attack must be guessed; the power model should be defined according to the characteristics of the attacked device, e.g., Hamming weight (HW) of the processed data for a microcontroller because of the existence of a precharged/predischarged bus in microcontrollers architecture. In case of a bad quality of the acquired power consumption, e.g., due to a noisy environment, bad measurement setup or cheap equipment, averaging can be applied by decrypting(encrypting) the same ciphertext(plaintext) repeatedly and calculating the mean of the corresponding traces to decrease the noise floor.

5 Our Proposed Attacks

In this section, we first specify the assumptions we have considered for a side-channel adversary in our proposed attacks. Afterwards, we review the side-channel vulnerabilities and information leakages which our specified adversary can recover considering the target microcontroller (AVR ATmega256). Taking the implementation profiles (defined in Section 3) into account different power analysis attacks are proposed in Section 5.2 to recover some secrets of the decryption algorithm. Finally, in Section 5.3 we discuss how to use the secrets recovered by means of the side-channel attacks to break the decryption scheme and reveal the system private key.

5.1 Adversary Model

In our proposed attacks we consider an adversary model:
The adversary knows what is public like $\hat{G}, t$. Also he knows the implementation platform (e.g., type of the microcontroller used), the implementation profile, i.e, complete source code of the decryption scheme (of course excluding memory

contents, precomputed values, and secret key materials). Also, he is able to select different ciphertexts and measure the power consumption during the decryption operation.

5.2 Possible Power Analysis Vulnerabilities

In order to investigate the vulnerability of the target implementation platform to power analysis attacks a measurement setup by means of an AVR ATmega256 microcontroller which is clocked by a 16MHz oscillator is developed. Power consumption of the target device is measured using a LeCroy WP715Zi 1.5GHz oscilloscope at a sampling rate of 10GS/s and by means of a differential probe which captures voltage drop of a 10Ω resistor at VDD (5V) path.

To check the dependency of power traces on operations, different instructions including arithmetic, load, and save operations are taken into account, and power consumption for each one for different operands are collected. In contrary to 8051-based or PIC microcontrollers, which need 16, 8, or 4 clock cycles to execute an operation, an AVR ATmega256 executes the instructions in 1 or 2 clock cycles[1]. Therefore, the power consumption pattern of different instructions are not so different from each other. As Figure 1 shows, though the instructions are not certainly recognizable, load instructions are detectable amongst others. As a result the adversary may be able to detect the execution paths by comparing the power traces. Note that as mentioned in Section 4 if the adversary is able to repeat the measurement for a certain input, averaging helps to reduce the noise and hence improve the execution path detection procedure.

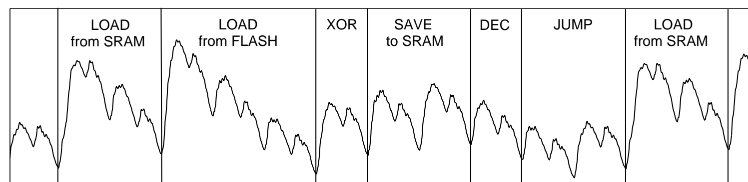

Fig. 1. A power consumption trace for different instructions

On the other hand, considering a fixed execution path, operand of instructions play a significant role in variety of power consumption values. As mentioned before, since the microcontrollers usually precharge/predischarge the bus lines, HW of the operands or HW of the results are proportional to power values. Figure 2 shows the dependency of power traces on the operands for XOR, LOAD, and SAVE instructions. Note that XOR instruction takes place on two registers, LOAD instruction loads an SRAM location to a specified register, and SAVE stores the content of a register back to the SRAM. According to Figure 2(c), HW of operands of SAVE instruction are more distinguishable in comparison to that

[1] Most of the arithmetic instructions in 1 clock cycle.

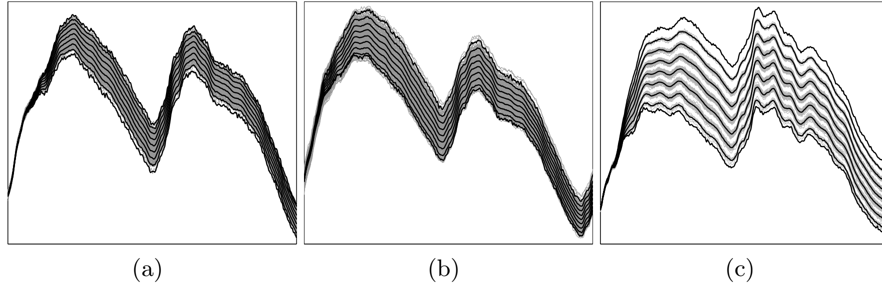

Fig. 2. Power consumption traces for different operands of (a) XOR, (b) LOAD, and SAVE instructions (all traces in gray and the averaged based on HWs in black)

of XOR and LOAD instructions. Therefore, according to the defined adversary model we suppose that the adversary considers only the leakage of the SAVE instructions. Now the question is "*How precisely the adversary can detect HW of the values stored by a SAVE instruction*?" It should be noted that a similar question has been answered in the case of a PIC microcontroller in [23] where the adversary (which fits to our defined adversary model in addition to profiling ability) has to profile the power traces in order to correctly detect the HWs. The same procedure can be performed on our implementation platform. However, in our defined adversary model the device under attack can be controlled by the attacker in order to repeat measurements as many as needed for the same input (ciphertext). Therefore, without profiling the attacker might be able to reach the correct HWs by means of averaging and probability distribution tests[2]. In contrary to an algebraic side-channel attack which needs all correct HW of the target bytes to perform a successful key recovery attack [23], as we describe later in Section 5.3 our proposed attack is still able to recover the secrets if the attacker guesses the HWs within a window around the correct HWs. Figure 3 presents success rate of HW detection for different scenarios. In the figure, the number of traces for the same target byte which are used in averaging is indicated by "avg". Further, "window" shows the size of a window which is defined around the correct HWs. As shown by Figure 3, to detect the correct HWs the adversary needs to repeat the measurements around 10 times, but defining a window by the size of 1 (i.e., correct HWs ± 1) leads to the success rate of 100% considering only one measurement.

Differential Power Analysis First, one may think that the best side-channel attack on implementation of McEliece decryption scheme would be a DPA to reveal the secret key. However, the input (ciphertext) is processed in a bitwise

[2] Probability distribution test here means to compare the probability distribution of the power values to the distribution of HW of random data in order to find the best match especially when highest (HW=8) or/and lowest (HW=0) is missing in measurements.

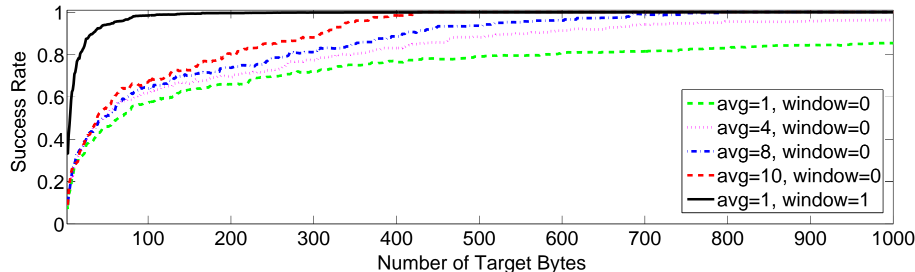

Fig. 3. Success rate of HW detection using the leakage of a SAVE instruction for different averaging and windowing parameters.

fashion, and in contrary to symmetric block ciphers the secret key does not contribute as a parameter of a computation. Moreover, power traces for different ciphertexts would not be aligned to each other based on the computations, and execution time of decryption also varies for different ciphertexts. As a consequence, it is not possible to perform a classical DPA attack on our target implementations.

SPA on Permutation Matrix Considering implementation profiles I and II (defined in Section 3) the first secret information which is used in decryption process is permutation matrix P. After permuting the ciphertext it is multiplied by matrix H^T. Since the multiplication of $\hat{c}$ and H^T can be efficiently realized by summing up those rows of H for which corresponding bit of $\hat{c}$ is "1" and skip all "0" bits, running time of multiplication depends on the number of "1"s (let say HW) of $\hat{c}$. As mentioned before the side-channel adversary would be able to detect the execution paths. If so, he can recover the content of $\hat{c}$ bit-by-bit by examining whether the summation is performed or not. However, HW of $\hat{c}$ is the same as HW of c, and only the bit locations are permuted. To recover the permutation matrix, the adversary can consider only the ciphertexts with HW=1 (2048 different ciphertexts in this case), and for each ciphertext finds the instant of time when the summation is performed (according to $\hat{c}$ bits). Sorting the time instants allows recovery whole of the permutation matrix. Figure 4 shows two power traces of start of decryption for two different ciphertexts. Obviously start of the summation is recognizable by visual inspection, but a general scheme (which is supposed to work independent of the implementation platform) would be similar to the scheme presented in [14]. That is, an arbitrary part of a trace can be considered as the reference pattern, and computing the cross correlation of the reference pattern and other power traces (for other ciphertexts with HW=1) reveals the positions in time when the summation takes place. Figure 5 presents two correlation vectors for the corresponding power traces of Figure 4. Note that to reduce the noise effect we have repeated the measurements and took the average over 10 traces for each ciphertext. Using this scheme for all ciphertexts with HW=1, permutation matrix is completely recovered.

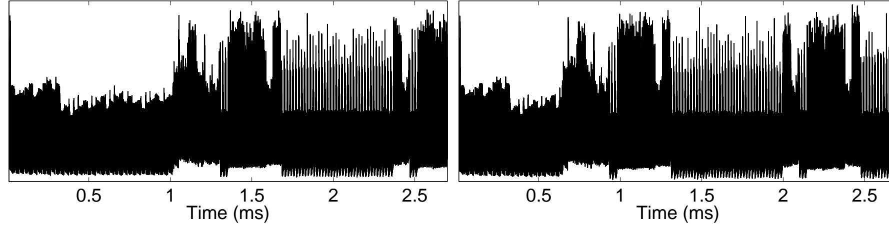

Fig. 4. Power traces of ciphertext (left) 0x0...01 and (right) 0x0...02.

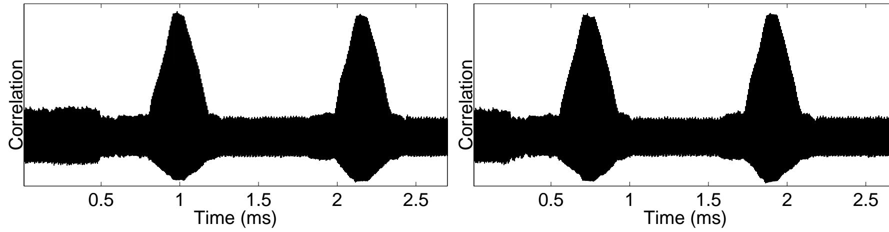

Fig. 5. Correlation vectors for ciphertexts (left) 0x0...01 and (right) 0x0...02.

SPA on Parity Check Matrix When implementation profiles III and IV are used, the permutation is not solely performed and hence the attack described above is not applicable. Therefore, the adversary has to take the multiplication process into account. Since in this case, execution path of multiplication does not depend on any secret, recovering the conditional branches (which only depend on ciphertext bits) would not help the attacker revealing the secrets. As a consequence the adversary has to try revealing the content of the parity check matrix H. To do so, as described before he may reach (or guess) HW of the processed (or saved) data. Similarly to the last scheme the attacker can chose all ciphertexts with HW=1 and guess the HW of elements of each column of matrix H separately. Since 27 11-bit elements of each column of H are saved efficiently in a byte-wise fashion in 38-byte chunks[3], and the adversary can only guess the HW of each byte, he can not certainly guess the HW of each 11-bit element of H. Therefore, the number of candidates for the HW of each 11-bit element is increased. As the result of this procedure, the adversary will have a set of candidates for each 11-bit element of parity matrix H at row i and column j as follows:

$$\hat{H}_{i,j} = \left\{ h \in \{0,1\}^{11} \ \mid \mathrm{HW}(h) \ = \ \text{the guessed HW by SPA } \pm \text{window} \right\}.$$

[3] Each 11-bit can be saved in 2 bytes, but it wastes the memory and also simplifies the attack procedure by dividing the HW of an 11-bit value to the HW of two 8- and 3-bit parts.

SPA on Goppa Polynomial If the attacker can follow the execution path after the matrix multiplication, he would be able to measure the power consumption during the computation of the syndrome polynomial inversion (step 2 of Algorithm 3). Since at the start of this computation the Goppa polynomial is loaded, e.g., from a nonvolatile memory to SRAM, similarly to the scheme explained above the adversary can predict HW of the transfered values, and hence make a list of candidates for each 11-bit element of the Goppa polynomial.

5.3 Gains of Power Analysis Vulnerabilities

This section discusses how to use the so far gathered information to perform a key recovery attack.

Attack I: Knowing the permutation matrix Given the permutation matrix P (which is recovered by means of an SPA), we are able to completely break the systemwith one additional assumption. We need to know the original support $\mathcal{L}$. In [10], Section $3.1 it is stated that $\mathcal{L}$ can be published without loss in security. Using the public key $\hat{G} = S * G * P$, we can easily recover $S * G$. Multiplication by a message with only a single "1" at position i gives us row $S[i]$ because G is considered to be in the systematic form. Therefore, by $(n - k)$ multiplications we can extract the scrambling matrix S and consequently G as well.

Now it is possible to recover the Goppa polynomial. According to Equation (1) we know that for a valid codeword (i.e., error free) the corresponding syndrome modulo $g(z)$ equals to zero. It means that the gcd of two different syndromes, which can now be computed by Equation (2) using $G' = S * G$ and the original support $\mathcal{L}$, equals $g(z)$ with high probability. In our experiments, it never took more than one gcd-computation to recover the correct Goppa polynomial.

From this point on, we have extracted all parameters of the McEliece system, and hence are able to decrypt every ciphertext. In order to verify the revealed secrets, we executed the key generation algorithm with the extracted parameters and retrieved exactly the same secret key as in the original setup.

Attack II: knowing parity check matrix Without knowing the original support $\mathcal{L}$, the attack described above is not applicable; moreover, in implementation profiles III and IV it is not possible to solely recover the permutation matrix. To overcome this problem we utilize the possible candidate lists $\hat{H}_{i,j}$ derived by an SPA attack. According to the structure of the parity check matrix H in Equation (3), every column is totally defined by elements α, $g(\alpha)$ and the coefficients of $g(z)$. We use this structure and the candidate lists in an exhaustive search. For every column $H[i]$ we randomly choose α_i and $g(\alpha_i)$ over all possible elements. These two elements are fixed for the entire column. Now we go recursively into the rows of column i. At every recursion level j we have to choose a random value for g_{t-j} and compute the actual value of $H[i][j]$ according to Equation (3). Only if this value is in the candidate list $\hat{H}_{i,j}$, we recursively call the search function for $H[i][j + 1]$. If a test fails, we remove the currently

selected element for g_{t-j} from the possible list and choose a new one. When the list gets empty, we return to one recursion level higher and try by a new element. Thereby we only go deeper into the search algorithm if our currently selected elements produce the values which are found in the corresponding candidate list. If the algorithm reaches $row[t+1]$, with $t = 27$ in our case, we have selected candidates for α_i, $g(\alpha_i)$, and all coefficients of the Goppa polynomial $g(z)$. Now we can check backwards whether $g(z)$ evaluates to $g(\alpha_i)$ at α_i. If so, we have found a candidate for the Goppa polynomial and for the first support element.

While the above described algorithm continues to search new elements, we can validate the current one. By choosing another column $H[i]$ and one of the remaining $n-1$ support elements, we can test in t trials whether the given value exists in the corresponding candidate list. On success we additionally found another support element. Repeating this step $n-1$ times reveals the order of the support $\mathcal{L}$ and verifies the Goppa polynomial. Column four in Table 1 shows the average number of false αs, that pass the first searched column for the right Goppa polynomial. However, these candidates are quickly sorted out by checking them against another column of H. For all remaining pairs $(\mathcal{L}, g(z))$ it is simply tested whether it is possible to decode an erroneous codeword.

Because a single column of H is sufficient for the first part of the attack, we could speed it up by selecting the column with the lowest number of candidates for the 27 positions. Depending on the actual matrix the number of candidates for a complete column varies between 1 000 and 25 000. It turns out that most often the column constructed by $\alpha = 0$ has the lowest number of candidates. So in a first try we always examine the column with lowest number of candidates with $\alpha = 0$ before iterating over other possibilities.

Also every information that one might know can speed up the attack. If, for example, it is known that a sparse Goppa polynomial is chosen, we can first test coefficient $g_i = 0$ before proceeding to other choices. For testing we generate a McEliece key from a sparse Goppa polynomial where only 4 coefficients are not zero. Table 1 shows the results for that key.

Even if the permutation matrix P is merged into the computation of H (implementation profiles III and IV) this attack reveals a permuted support $\mathcal{L}_P$, which generates a parity check matrix capable of decoding the original ciphertext c. As a result, although merging P and H is reasonable from a performance point of view, this eases our proposed attack.

Attack III: Improving Attack II Considering the fact mentioned at the end of Section 5.2 knowing some information about the coefficients of $g(z)$ dramatically reduces the number of elements to be tested on every recursion level. The use of additional information, here the HW of coefficients of $g(z)$, significantly speeds up the attack, as shown in Table 2.

As mentioned in the previous section, Table 1 shows the results for a sparse Goppa polynomial. These result were achieved using a workstation PC equipped by two Xeon E5345 CPUs and 16 GByte RAM and gcc-4.4 together with OpenMP-3.0. The results for a full random Goppa polynomial are given in Table 2.

Table 1. Runtime of the Search Algorithm for sparse Goppa polynomial

Window Size H	**Window Size** $g(z)$	$\#g(z)$	$\#\ \alpha$	**CPU Time**
0	X	$> 10^6$	112	115 hours
1	X	$> 2^{32}$	$> 2^{32}$	150 years
0	0	3610	68	< 1 sec
1	0	112527	98	10 sec
0	1	793898	54	186 min
1	1	$> 10^6$	112	71 days

Table 2. Runtime of the Search Algorithm for full random Goppa polynomial

Window Size H	**Window Size** $g(z)$	$\#g(z)$	$\#\ \alpha$	**CPU Time**
0	X	$> 10^6$	52	90 hours
1	X	$> 2^{32}$	$> 2^{32}$	*impossible*
0	0	4300	50	69 min
1	0	101230	37	21 hours
0	1	$> 2^{32}$	$> 2^{32}$	26 days
1	1	$> 2^{32}$	$> 2^{32}$	5 years

In this table a window size of X means that we do not use the information about the Goppa polynomial. Instead, we iterate over all possibilities. $\#g(z)$ denotes the number of Goppa polynomials found until the correct one is hit, and $\#\ \alpha$ indicates how many wrong elements fulfil even the first validation round. The column *CPU Time* is the time for a single CPU core.

Note that the values in the second and last row of each table are only estimates. They are based on the progress of the search in around 2 weeks and on the knowledge of the right values. The *impossible* means, that there was only little progress and the estimate varied by hundreds of years.

Also it should be investigated whether the additional information from the side-channel attacks can improve one of the already known attacks, e.g., [2, 16, 17, 31]. The information gathered by means of side-channels ought to be useful since it downsizes the number of possibilities.

6 Countermeasures

Since the multiplication of the permuted ciphertext and parity check matrix H^T is efficiently implementing by summing up (XORing) some H rows which have "1" as the corresponding permuted ciphertext, the order of checking/XORing H rows can be changed arbitrarily. Since we have supposed that the attacker (partially) knows the program code, any fix change on the execution path, e.g., changing the order of summing up the H rows would not help to counteract our first attack (SPA on permutation matrix explained in Section 5.2). However, one can change the order of checking/XORing randomly for every ciphertext, and

hence the execution path for a ciphertext in different instances of time will be different. Therefore, the adversary (which is not able to detect the random value and the selected order of computation) can not recover the permutation matrix. Note that as mentioned before if the permutation is not merely performed (e.g., in implementation profiles III and IV) our first attack is inherently defeated.

Defeating our second attack (SPA on parity check matrix explained in Section 5.2) is not as easy as that of the first attack. One may consider changing randomly the order of checking the H rows, which is described above, as a countermeasure against the second attack as well. According to the attack scenario the adversary examines the power traces for the ciphertexts with HW=1; then, by means of pattern matching techniques he would be able to detect at which instance of time the desired XOR operations (on the corresponding row of H) is performed. As a result, randomly changing the order to computations does not help to defeat the second attack. An alternative would be to randomly execute dummy instructions[4]. Though it leads to increasing the run time which is an important parameter for post quantum cryptosystems especially for software implementations, it extremely hardens our proposed attacks. A boolean masking scheme may also provide robustness against our attacks. A simple way would be to randomly fill the memory location which stores the result of XORing H rows before start of the multiplication (between the permuted ciphertext and the parity check matrix), and XORing the final results by the same start value. This avoids predicting HW of H elements if the attacker considers only the leakage of the SAVE instructions. However, if he can use the leakage of LOAD instructions (those which load H rows), this scheme does not help to counteract the attacks. One can make a randomly generated mask matrix as big as H, and save the masked matrix. Since in order to avoid the effect of the masking after multiplication it is needed to repeat the same procedure (multiplication) using the mask matrix, this scheme doubles the run time (for multiplication) and the area (for saving the mask matrix) as well though it definitely prevents our proposed attacks. As a result designing a masking scheme which is adopted to the limitations of our implementation platform is considered as a future work.

7 Conclusions

In this paper, we presented the first practical power analysis attacks on different implementations of the McEliece public-key scheme which use an 8-bit general-purpose AVR microprocessor to realize the cipher decryption. Since we believe that with growing memory of embedded systems and future optimizations McEliece can be developed as a quantum computer-resistant replacement for RSA and ECC, vulnerability and robustness of McEliece implementations in the presence of side-channel attacks should be addressed before its widespreading into pervasive applications and devices which are under control of the side-channel adversaries. Further, to defeat the described vulnerabilities we intro-

[4] In our implementation platform it can be done by a random timer interrupt which runs a random amount of dummy instructions.

duced and discussed possible countermeasures which seem not to be perfect because of their high time and memory overheads. As a result, designing a suitable countermeasure which fits to the available resources of low-cost general-purpose microprocessors and provides a reasonable level of security against side-channel attacks is considered as a future work. This work shows clearly that every part of the secret key materials namely the support $\mathcal{L}$, the Goppa polynomial $g(z)$, the permutation P and every (precomputed) part of the parity check matrix H have to be well protected.

References

1. E. R. Berlekamp. Goppa Codes. *IEEE Trans. on Information Theory*, 19(3):590–592, 1973.
2. D. J. Bernstein, T. Lange, and C. Peters. Attacking and Defending the McEliece Cryptosystem. In *Post-Quantum Cryptography - PQCrypto 2008*, volume 5299 of *LNCS*, pages 31–46. Springer, 2008. Also available on `http://eprint.iacr.org/2008/318`.
3. B. Biswas and N. Sendrier. McEliece Cryptosystem Implementation: Theory and Practice. In *Post-Quantum Cryptography - PQCrypto 2008*, volume 5299 of *LNCS*, pages 47–62. Springer, 2008.
4. A. Bogdanov, I. Kizhvatov, and A. Pyshkin. Algebraic Methods in Side-Channel Collision Attacks and Practical Collision Detection. In *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *LNCS*, pages 251–265. Springer, 2008.
5. E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.
6. P.-L. Cayrel and P. Dusart. Fault Injection's Sensitivity of the McEliece PKC. 2009. Available on `http://www.cayrel.net/IMG/pdf/Fault_injection_s_sensitivity_of_the_McEliece_PKC.pdf`.
7. B. den Boer, K. Lemke, and G. Wicke. A DPA Attack against the Modular Reduction within a CRT Implementation of RSA. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *LNCS*, pages 228–243. Springer, 2002.
8. T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar. MicroEliece: McEliece for Embedded Devices. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *LNCS*, pages 49–64. Springer, 2009.
9. T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme. In *Advances in Cryptology - CRYPTO 2008*, volume 5157 of *LNCS*, pages 203–220. Springer, 2008.
10. D. Engelbert, R. Overbeck, and A. Schmidt. A Summary of McEliece-Type Cryptosystems and their Security. *Journal of Mathematical Cryptology*, 1(2):151–199, 2006. Also available on `http://eprint.iacr.org/2006/162`.
11. B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. Mutual Information Analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2008*, volume 5154 of *LNCS*, pages 426–442. Springer, 2008.
12. S. Hoerder. Explicit Computational Aspects of McEliece Encryption Scheme. Master's thesis, Ruhr University Bochum, Germany, 2009.

13. T. Howenga. Efficient Implementation of the McEliece Cryptosystem on Graphics Processing Units. Master's thesis, Ruhr-University Bochum, Germany, 2009.
14. M. Kasper, T. Kasper, A. Moradi, and C. Paar. Breaking KeeLoq in a Flash. In *Progress in Cryptography - AFRICACRYPT 2009*, volume 5580 of *LNCS*, pages 402–419. Springer, 2009.
15. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology - CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
16. P. J. Lee and E. F. Brickell. An Observation on the Security of McEliece's Public-Key Cryptosystem. In *Advances in Cryptology - EUROCRYPT 1988*, volume 330 of *LNCS*, pages 275–280. Springer, 1988.
17. J. S. Leon. A Probabilistic Algorithm for Computing Minimum Weights of Large Error-Correcting Codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, 1988.
18. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, Jan. 1978.
19. T. S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *LNCS*, pages 238–251. Springer, 2000.
20. E. Oswald. Enhancing Simple Power-Analysis Attacks on Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *LNCS*, pages 82–97. Springer, 2002.
21. E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In *The Cryptographers' Track of the RSA Conference - CT-RSA 2006*, volume 3860 of *LNCS*, pages 192–207. Springer, 2006.
22. N. Patterson. The Algebraic Decoding of Goppa Codes. *IEEE Transactions on Information Theory*, 21:203–207, 1975.
23. M. Renauld, F.-X. Standaert, and N. Veyrat-Charvillon. Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *LNCS*, pages 97–111. Springer, 2009.
24. K. Schramm, G. Leander, P. Felke, and C. Paar. A Collision-Attack on AES: Combining Side Channel- and Differential-Attack. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *LNCS*, pages 163–175. Springer, 2004.
25. K. Schramm and C. Paar. Higher Order Masking of the AES. In *The Cryptographers' Track of the RSA Conference - CT-RSA 2006*, volume 3860 of *LNCS*, pages 208–225. Springer, 2006.
26. K. Schramm, T. J. Wollinger, and C. Paar. A New Class of Collision Attacks and Its Application to DES. In *Fast Software Encryption - FSE 2003*, volume 2887 of *LNCS*, pages 206–222. Springer, 2003.
27. A. Shoufan, F. Strenzke, H. G. Molter, and M. Stoettinger. A Timing Attack Against Patterson Algorithm in the McEliece PKC. In *International Conference on Information Security and Cryptology - ICISC 2009*, LNCS. Springer, 2009. to appear.
28. A. Shoufan, T. Wink, G. Molter, S. Huss, and F. Strentzke. A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms. In *Application-specific Systems, Architectures and Processors - ASAP 2009*, pages 98–105. IEEE Computer Society, 2009.

29. J. H. Silverman and W. Whyte. Timing Attacks on NTRUEncrypt Via Variation in the Number of Hash Calls. In *The Cryptographers' Track of the RSA Conference - CT-RSA 2007*, volume 4377 of *LNCS*, pages 208–224. Springer, 2007.
30. F.-X. Standaert, S. B. Örs, J.-J. Quisquater, and B. Preneel. Power Analysis Attacks Against FPGA Implementations of the DES. In *Field Programmable Logic and Application - FPL 2004*, volume 3203 of *LNCS*, pages 84–94. Springer, 2004.
31. J. Stern. A Method for Finding Codewords of Small Weight. In *Coding Theory and Applications*, volume 388 of *LNCS*, pages 106–113. Springer, 1989.
32. F. Strenzke, E. Tews, H. G. Molter, R. Overbeck, and A. Shoufan. Side Channels in the McEliece PKC. In *Post-Quantum Cryptography - PQCrypto 2008*, volume 5299 of *LNCS*, pages 216–229. Springer, 2008.
33. H. C. van Tilborg. *Fundamentals of Cryptology.* Kluwer Academic Publishers, 2000.
34. N. V. Vizev. Side Channel Attacks on NTRUEncrypt. Bachelor's thesis, Technical University of Darmstadt, Germany, 2007. Available on `http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Nikolay_Vizev.bachelor.pdf`.

Appendix

Algorithm 3 Decoding Goppa Codes

Require: Received codeword r with up to t errors, inverse generator matrix iG
Ensure: Recovered message $\hat{m}$
1: Compute syndrome $Syn(z)$ for codeword r
2: $T(z) \leftarrow Syn(z)^{-1} \mod g(z)$
3: **if** $T(z) = z$ **then**
4: $\quad \sigma(z) \leftarrow z$
5: **else**
6: $\quad R(z) \leftarrow \sqrt{T(z) + z}$
7: $\quad$ Compute $a(z)$ and $b(z)$ with $a(z) \equiv b(z) \cdot R(z) \mod g(z)$
8: $\quad \sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
9: **end if**
10: Determine roots of $\sigma(z)$ and correct errors in r which results in $\hat{r}$
11: $\hat{m} \leftarrow \hat{r} \cdot iG$ {Map r_{cor} to $\hat{m}$}
12: **return** $\hat{m}$